Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Кафедра системного программирования

Мисонижник Александр Владимирович

Композициональные частично определенные типы в символьном интерпретаторе для платформы .NET Framework

Выпускная квалификационная работа

Научный руководитель:
ст. преп. Кириленко Я. А.

Рецензент:
Принстонский университет
постдок
кандидат наук, Федюкович Г. Г.

Санкт-Петербург
2018

SAINT PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
System Programming

Aleksandr Misonizhnik

Compositional Partially Defined Types in Symbolic Interpreter of the .NET Framework

Bachelor's Thesis

Scientific supervisor:
Senior lecturer Iakov Kirilienko

Reviewer:
Princeton University
Postdoctoral Research Associate
Ph. D. Grigory Fedyukovich

Saint Petersburg
2018

Содержание

Введение

В области верификации есть методы, которые используют технику *символьного исполнения* [2, 12, 13] для сведения кода к языку логики первого порядка и *решатели дизъюнктов Хорна* [8, 9, 14] для их разрешения. Одна из главных задач, решаемых символьным интерпретатором — исследование веток выполнения кода на неопределенных входных данных.

Во время символьного исполнения .NET-кода появляется проблема, порождаемая неопределенностью данных, а именно неопределенность динамических типов объектов. Один из самых простых примеров, демонстрирующих эту проблему — функция `Object.Equals(Object)`. Вне контекста конкретного вызова невозможно понять, какой именно объект будет передан в качестве параметра этого метода, в частности, неизвестно какой у объекта будет тип. Ситуация осложняется наличием *обобщений* (*generics*) с ограничениями на типовые параметры, а также *вариантностью* для обобщенных интерфейсов. Не стоит забывать и о сложности системы типов .NET. В частности, решение должно естественным образом масштабироваться на массивы символьных размерностей, делегаты, unsafe-указатели и т. д.

Одна из главных проблем, от которой страдают символьные интерпретаторы — проблема *взрыва путей исполнения*: количество путей исполнения программы в общем случае зависит экспоненциально от ее размера[1]. Один из подходов, направленных на решение этой проблемы — это *композиционное символьное исполнение* [6], которое позволяет переиспользовать уже проинтерпретированные участки кода во время символьного исполнения. В случае с динамическими типами объектов композиционность заключается в возможности получить корректный результат интерпретации кода, при условии того, что типы объектов могут быть конкретизированы, без повторного исполнения.

`V#` — это проект, направленный на создание неограничиваемого верификатора для платформы .NET. Помимо огромной важности для самого проекта `V#`, эта задача может представлять интерес для акаде-

мического сообщества. К примеру, существует довольно большое количество работ, посвященных статическому анализу языка Java (одна из последних обзорных статей на эту тему является [16]). Существуют и работы на тему символьного исполнения в условиях неопределенности типа объектов, например, [10]. Стоит отметить также, что в наиболее известных state-of-art-интерпретаторах (таких как Java PathFinder [7] и Pex [21]), символьное исполнение кода со сложными типами поддержано в весьма ограниченном виде. В частности, не один из этих инструментов не поддерживает исполнение кода с открытыми типами в полном объёме, хотя возможность получить корректный результат символьной интерпретации кода с открытыми типами в сочетании с композициональностью позволяет сильно повысить эффективность исполнения.

1. Постановка задачи

Целью работы является поддержка композиционного символьного исполнения .NET-кода в условиях неопределенности динамических типов объектов. В контексте данной работы были поставлены следующие задачи.

- Спроектировать архитектуру подсистемы, отвечающей за интерпретацию типов.
- Реализовать символьное исполнение функций с учетом открытых типов и полиморфизма, учитывая необходимость композициональности исполнения.
- Разработать и реализовать алгоритм решения систем ограничений на типы.
- Провести тестирование функциональности реализованной подсистемы.

2. Обзор

Причиной возникновения неопределенности динамических типов объектов является полиморфизм, который основан на отношении подтипирования. Возникает естественная необходимость отобразить это отношение в результате символьного исполнения. Для этого надо разобраться в системе типов языка CIL. Язык C# явно отображает систему типов CIL, так что можно ограничится рассмотрением системы типов языка C#. Но сначала надо описать некоторые важные понятия для пониманию сути символьного исполнения.

2.1. Символьное исполнение

В проекте `V#` техника композиционального символьного исполнения используется для сведения MSIL-кода к языку логики первого порядка. Эта техника позволяет моделировать исполнение кода, при котором часть входных данных остаётся неопределенной. Основными понятиями для символьного исполнения являются символьное состояние и условие реализуемости пути (path condition). Символьное состояние описывает множество переменных и их значений, которые могут быть символьными. Оно состоит из объединения состояний, полученных в разных путях исполнения. Каждое из таких состояний хранит условие реализуемости пути, при истинности которого это состояние будет релевантным. Условие реализуемости пути преобразуется в формулу логики первого порядка, а значит можно попробовать проверить его выполнимость, отдав формулу автоматическому решателю формул.

Композициональность позволяет переиспользовать результаты уже исполненных участков кода. Например, если во время символьного исполнения функции F надо будет исполнить вызов другой функции G, которая уже была исполнена, нужно будет конкретизировать в результат исполнения функции G, используя текущее символьное состояние, полученное исполнением функции F. После этого останется лишь пересчитать условия реализуемости пути, чтобы избавиться от недостижимых путей исполнения. Такой подход значительно увеличивает эф-

фективность символьного исполнения.

2.2. Система типов

В объектно-ориентированных языках, и, в частности, в языке C#, основой системы типов является номинальная система типов. Это означает, что отношение подтипирования явно определяется в коде программы. В частности, в C# номинальное подтипирование задаётся через отношение наследования. Но также в системе типов C# присутствуют особенности структурной системы типов, которые выражаются наличием вариантности по типовому параметру обобщенных типов. Это значит, что отношение подтипирования между такими типами задаётся неявно через типовые аргументы [11].

Все типы, за исключением типов unsafe-указателей, являются подтипами типа `Object`. Эти типы можно разбить на две группы: типы классов и типы интерфейсов. Для первой группы запрещено множественное наследование, для второй разрешено. Также разрешено множественное наследование интерфейсов классами.

И классы, и интерфейсы могут быть обобщенными. Обобщенность также является причиной появление неопределенности динамических типов объектов. Кроме того, типовые параметры обобщенных типов могут задавать очень сложные ограничения на структуру подтипирования.

Из-за большой сложности системы типов возникает желание отстраниться от всех «тонкостей» отношения подтипирования и естественным образом разветвлять исполнение всякий раз, когда необходимо будет уточнить динамический тип объекта. Такого подхода придерживаются при динамическом символьном исполнении, в частности в интерпретаторе Pex [21]. К сожалению, такой подход порождает слишком много веток исполнения, усугубляя проблему взрыва путей исполнения.

К счастью, существует работа, которая формально описывает все «тонкости» номинальных систем типов с вариантностью, и систему типов C# в частности [11]. Из работы следует, что отношение подтипи-

рования между типами, не содержащими типовых переменных, разрешимо.

2.3. Существующие подходы

В работе [3] проведено подробное сравнение нескольких инструментов для генерации тестов, которые основаны на статическом анализе кода. В частности, проведено сравнение на тестах направленных на анализ кода с обобщенными типами. Среди них есть state-of-art-интерпретаторы Java PathFinder и Pex, но даже они показали не самые лучшие результаты при исполнении кода содержащего обобщенные типы. Большего всего отличился инструмент EvoSuite [5], однако, в основе его анализа лежит генетический алгоритм, а не символьное исполнение, так что этот инструмент не представляет интереса в рамках этой работы.

Важной частью символьной интерпретации типов является кодирование ограничений на эти типы в логику. Существует несколько работ, посвященных этой проблеме, к примеру, [17, 20]. В работе [17] подробно описывается кодирование обобщенных типов в логику с помощью эмуляции алгебраических типов данных логикой первого порядка. Но в ней никак не затрагивается кодирование отношения подтипирования. Напротив, в работе [20] предложен способ аксиоматизации логики первого порядка с номинальной типизацией, в которой есть аналоги операторов, связанных с типами (`is`, `as` и оператор преобразования типа). Однако, её недостаточно, чтобы закодировать систему типов .NET с её структурными элементами.

3. Архитектура подсистемы интерпретации типов

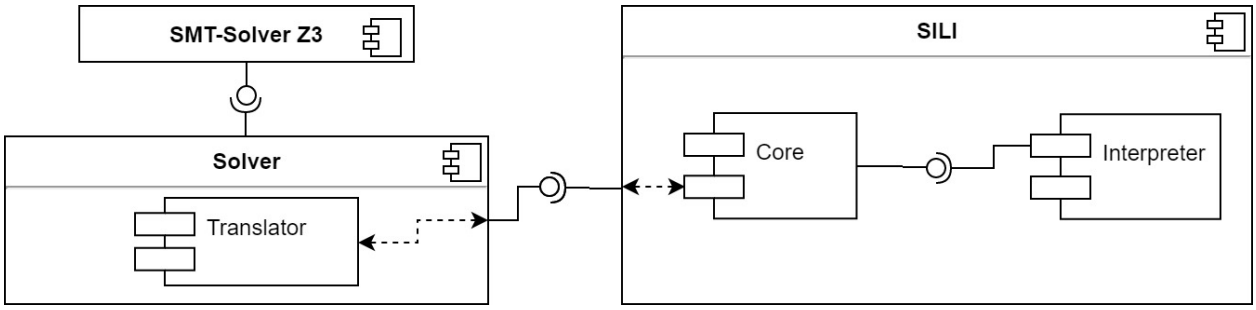

Рис. 1: Архитектура подсистемы символьного интерпретатора.

Архитектура подсистемы должна быть спроектирована таким образом, чтобы соответствовать архитектуре символьного интерпретатора.

На Диаграмме 1 описана архитектура символьного интерпретатора и подсистемы, отвечающей за интерпретацию типов, в проекте `V#`. Компонента SILI, непосредственно отвечающая за символьную интерпретацию, реализована на языке Ядро символьного исполнения имеет внешний интерфейс, который содержит в себе основные функции для управления исполнением кода, а интерпретатор производит интерпретацию кода согласно потоку управления.

Интерпретатор разветвляет исполнение всякий раз, когда интерпретация может пойти по разным путям. Разветвление происходит по условию, которое добавляется в условие реализцемости пути. Для того, чтобы не исполнять заведомо недостижимые пути, полученное условие пути исполнения преобразуется в формат, который понимает SMT-решатель формул и передаётся решателю, чтобы проверить выполнимость формулы. Если формула не выполнима, можно далее не исполнять код с текущим условием исполнения пути, так как этот путь исполнения недостижим.

Для того, чтобы поддержать символьное исполнение кода с учетом полиморфизма и открытых типов, нужно реализовать в ядре возможность символьно представлять неопределенность динамических типов объектов и ограничения на них. Также, в интерпретаторе нужно реализовать корректное символьное исполнение кода, непосредственно за-

висящего от типов:

- вызов методов от аргументов с неопределенным типом;
- вызов операторов is, as, преобразования типа;
- вызов обобщенных методов.

.

В результате, в условии реализуемости пути будут появляться ограничения на типы, которые нужно уметь решать, чтобы определить выполнимость условия реализуемости пути. Так как задача выполнимости условия решается с помощью преобразования условия пути в формулу логики первого порядка, надо разработать и реализовать алгоритм, который сведёт задачу решения ограничений на типы к задаче выполнимости формулы первого порядка.

На Диаграмме 1 видно, что процесс реализации необходимой функциональности был декомпозирован на задачи реализации частей компонент символьного интерпретатора.

4. Композициональное символьное исполнение с типами

Реализация функциональности в компоненте непосредственно отвечающей за символьную интерпретацию разделяется на две задачи: реализация символьного представления неопределенности динамических типов и реализация композиционального символьного исполнения с учетом неопределенности типов.

4.1. Частично определенные типы

Неопределенность динамического типа в ядре интерпретатора представляется отдельной сущностью — *частично определенным типом*. Частично определенный тип является аналогом символьной переменной для типов. Таким образом любой объект, тип которого станет известен только во время конкретного исполнения, будет иметь частично определенный тип, который отображает пространство всех возможных типов. Ограничения на тип представляются в виде символьных булевых термов. Среди них есть двухместный предикатный символ отношения подтипирования и несколько одноместных предикатных символов, отображающих свойства типов: является ли тип интерфейсом, классом, структурой, есть ли у типа конструктор без параметров. Символьное исполнение операторов as, is и преобразования типа может быть выражено с помощью ограничений на типы:

- результатом исполнения оператора is является символьный булев терм, состоящий из предикатного символа отношения подтипирования;
- результат исполнения оператора as получается путем разветвления исполнения по условию подтипирования типа объекта к приводимому типу: если условие истинно, то результатом будет ссылка на объект с измененным типом, иначе результатом будет `null`;
- результат исполнения оператора преобразования типа аналогичен

результату исполнения оператора as, за одним исключением: если условие ложно, исполнить бросание исключения недопустимого преобразования типа.

Также разветвление исполнения по условию подтипирования будет производиться всякий раз, когда от типа зависит результат исполнения: вызов виртуальных и абстрактных методов, вызов делегатов.

4.2. Подстановка типов

Так как непопределенность динамических типов стала явно представляться в символьном интерпретаторе отдельной сущностью, символьная интерпретация кода с частично определенными типами ничем не отличается от интерпретации кода с конкретными типами. Однако остается вопрос о реализации композициональности исполнения с учетом типов.

Результатом символьного исполнения является символьное состояние, в котором вся необходимая информация хранится в виде структурированного набора символьных термов, которые внутри себя содержат информацию о типах. Таким образом, взятие композиции, в случае с частично определенными типам, является применением подстановки конкретизированных типов. Для удобства, разделим частично определенные типы на два вида: *явные* частично определенные типы, отображающие неопределенность типовых переменных, и *неявные*, отображающие неопределенность полиморфных типов.

Реализация подстановки в случае с явными типами потребовала расширения определения символьного состояния словарем, в котором ключами являются явные типовые переменные, а значениями — их конкретизированные версии. Состояние словаря зависит от текущей области исполнения метода: так, каждый раз при исполнении вызова обобщенного метода, свойства или чтения поля с обобщенным типом, в словарь будут добавляться записи, конкретизирующие явный тип, во время исполнения будет производиться подстановка явных типов на их конкретизированные версии, а после завершения исполнения кода с обобщен-

ными типами записи будут удаляться.

В случае с неявными типами всё становится немного сложнее: тип конкретизируется не через явное применение типовых аргументов, а неявно, через конкретизацию объекта, имеющего этот тип. В этом случае, для реализации композициональности, надо расширить определение неявного типа, добавив в него информацию об объекте, типом которого он является. Таким образом, при взятии композиции объект, у которого был неявный тип, конкретизируется, а при взятии композиции для неявного типа можно будет посмотреть текущий тип конкретизированного объекта, и применить подстановку с его конкретизированным типом.

Рассмотренные случаи полностью описывают операцию взятия композиции в случае с частично определенными типами. После применения композиции остаётся лишь пересчитать выполнимость условий реализации пути, чтобы отсечь нерелевантные части символьного состояния. В следующей главе будет описано решение задачи выполнимости систем ограничений на типы.

5. Формализация системы типов .NET

Для того, чтобы разработать алгоритм решения систем ограничений на типы, будет удобно формализовать систему типов .NET.

Типы (обозначим их как T, U, V и W) можно разделить на два вида: типовые переменные X, Y и Z и сконструированные типы $C\texttt{<}\overline{T}\texttt{>}$, где C — это конструктор типа, а $\overline{T}$ — упорядоченный список аргументов типов. *Закрытыми* будем называть типы, которые в листьях дерева конструкторов не содержат типовые переменные. *Открытыми* назовём все типы, которые не являются закрытыми. *Высотой* типа будем называть высоту дерева его конструкторов. Обозначим высоту функцией $height$.

Можно считать, что все типы являются подтипами `System.Object`: типы указателей здесь рассматриваться не будут из-за тривиальности их подтипирования.

Номинальность системы типов означает, что отношение подтипирования явно определяется в коде программы и может быть представлено в виде конечной *таблицы классов*. Каждая запись в таблице классов для системы типов .NET имеет следующий вид:

$$C^{*}\texttt{<}\overline{X}\texttt{>} <:: T_1, \ldots, T_n$$

Для каждого типового параметра имеется запись, которая ограничивает его:

$$C\#i^{*} <: U_1, \ldots, U_k \tag{1}$$

Без потери общности можно считать, что в таблице классов нет двух типовых переменных с одним названием (иначе переименуем их).

Левая часть каждой записи в таблице классов содержит аннотацию типа, помещаемую вместо *. Аннотация указывает, является ли тип интерфейсом (в таком случае будет записана буква I), типом значения (V), ссылочным типом (R), классом, массивом или делегатом (C), запечатанным типом (аннотация взята в квадратную рамку, например, $\boxed{C}$) и имеется ли у типа конструктор без аргументов (в таком случае

около буквы записана пара скобок).

Например, для `System.Object`, `System.IComparable` и структуры `S` (не реализующей ни одного интерфейса), записи будут выглядеть следующим образом:

$$\begin{aligned}
&\texttt{System.Object}^{\,C()} && <:: \\
&\texttt{System.IComparable}^{\,I} && <:: \texttt{System.Object} \\
&\texttt{S}^{\boxed{V()}} && <:: \texttt{System.ValueType}
\end{aligned}$$

Символ $<::$ обозначает отношение номинального подтипирования. Доопределим также $C\texttt{<}\overline{U}\texttt{>} <:: [\overline{U}/\overline{X}]T_i$. Через $<::^+$ обозначим его транзитивное замыкание. Отношение $<::^+$ отражает номинальность системы типов, но его не достаточно, чтобы выразить структурные элементы. Для отражения вариантности, потребуем от записей таблицы классов иметь вид

$$C^*\texttt{<}v_1X_1, \ldots, v_mX_m\texttt{>} <:: \ T_1, \ldots, T_n,$$

где v_i определяет вариантность по i-тому типовому параметру и может принимать следующие значения: $\circ$ (инвариантность), $+$ (ковариантность), $-$ (контравариантность). Положим $C\#i \stackrel{\text{def}}{=} X_i$ и $var(C\#i) \stackrel{\text{def}}{=} v_i$.

Заметим, что в записях вида (1) содержится символ подтипирования $<:$ вместо символа номинального подтипирования. Отношение подтипирования в .NET будет строго определено ниже.

Наложим ограничения на таблицу классов:

1. отношение $<::^+$ должно быть ацикличным;

2. записи в таблице должны быть корректными относительно вариантности: например, запрещено

$$\begin{aligned}
&B^{\,I}\texttt{<}-X\texttt{>} && <:: \ \ldots \\
&A^{\,I}\texttt{<}+X\texttt{>} && <:: B\texttt{<}X\texttt{>}
\end{aligned}$$

3. запечатанному типу (в т.ч. делегату или массиву) разрешено по-

являться в правой части записи только в качестве типового аргумента;

4. в правой части любой записи может стоять максимум один тип с аннотацией C (множественное наследование запрещено);

5. аннотации не должны противоречить естественным ограничениям .NET (интерфейс не может быть запечатанным или иметь конструктор без аргументов, типы значений обязательно запечатаны и имеют конструктор без аргументов и т.д.).

Определение 1. *Расширением* таблицы классов CT называется таблица классов, в которой содержатся все записи из CT, а все левые части прочих записей содержат лишь конструкторы и их типовые переменные, не входящие ни в одну из левых частей CT.

Наконец, можно определить отношение подтипирования с учетом вариантности.

Определение 2. Отношение подтипирования для закрытых типов $<:$ задается следующими правилами:

$$(\text{Var})\frac{\text{for each } i \quad T_i <:_{var(C\#i)} U_i}{C\texttt{<}\overline{T}\texttt{>} <: C\texttt{<}\overline{U}\texttt{>}}$$

$$(\text{Super})\frac{C\texttt{<}\overline{X}\texttt{>} <:: V \quad [\overline{T}/\overline{X}]V <: D\texttt{<}\overline{U}\texttt{>}}{C\texttt{<}\overline{T}\texttt{>} <: D\texttt{<}\overline{U}\texttt{>}} C \neq D$$

$$\frac{T <: U}{T <:_+ U} \qquad \frac{}{T <:_\circ T} \qquad \frac{U <: T}{T <:_- U}$$

Предложение 1. Отношение подтипирования $<:$ разрешимо для закрытых типов, при условии, что таблица классов будет *нерасширяющийся*.

За определением нерасширяемости таблицы классов и доказательством Предложения 1 читатель отсылается к работе [11]. Из этого следует разрешимость подтипирования для закрытых типов .NET, т.к. по спецификации любая расширяемая таблица классов отклоняется средой выполнения [4].

Другими словами, работа [11] доказывает существование алгоритма, принимающего на вход любую таблицу классов (прошедшую валидацию среды выполнения .NET) и утверждение вида $T <: U$, где T и U — *закрытые* типы. Данная работа решает аналогичную задачу, но для *открытых* типов.

Очевидно, вопрос $T <: U$ в случае открытых типов сам по себе не имеет смысла: уже для двух типовых переменных на вопрос $X <: Y$ можно дать как утвердительный, так и отрицательный ответ; такая постановка имеет смысл лишь при связывании типовых переменных кванторами. Естественной, поэтому, будет формулировка задачи в терминах логики первого порядка.

6. Алгоритм решения систем ограничений на типы

На протяжении всей главы предполагается, что имеется и зафиксирована таблица классов CT.

Рассмотрим язык первого порядка $\mathcal{L}_t$ над сигнатурой $(\mathcal{F}_t, \mathcal{P}_t)$. Здесь

- $\mathcal{F}_t$ — множество функциональных символов, отождествляемое с множеством конструкторов типов. Для удобства применение функционального символа C к аргументам $\overline{T}$ будет по-прежнему записываться как $C\texttt{<}\overline{T}\texttt{>}$.

- $\mathcal{P}_t = \{<:\}$, где $<:$ является двухместным предикатным символом и будет записываться в инфиксном стиле. Для удобства, формулы вида $\neg(T <: U)$ и $\neg(T = U)$ будут записываться как $T \not<: U$ и $T \neq U$.

Данная глава описывает алгоритм решения следующей задачи. По натуральному числу h и данной формуле $\phi_t \in \mathcal{L}_t$ без кванторов необходимо построить такую модель $\mathcal{M}_t = (\mathcal{D}_t, \sigma_t)$, что $\mathcal{M}_t \models cl_\exists(\phi_t)$, либо доказать, что такой модели не существует. Однако на $\mathcal{M}_t$ должен накладываться набор ограничений;

- носитель $\mathcal{D}_t$ должен быть подмножеством множества замкнутых типов;

- для каждого $C \in \mathcal{F}_t$, $\sigma_t(C)(\overline{T})$ должен конструировать замкнутый тип $C\texttt{<}\overline{T}\texttt{>}$;

- $\sigma_t(<:) \subseteq \mathcal{D}_t \times \mathcal{D}_t$ — отношение подтипирования, заданное некоторым расширением таблицы классов CT, а $cl_\exists(\phi)$ — экзистенциальное замыкание формулы ϕ_t (т.е. формула $\exists x_1, \ldots, x_n.\phi$, где $x_1, \ldots, x_n$ — свободные переменные в ϕ);

- val_t — оценка переменных $fv(\phi_t)$ и

$$\max_{T \in val_t(fv(\phi_t))} height(T) \leqslant h$$

- семантика равенства стандартна.

Основным препятствием для применения решателей логики первого порядка к данной задаче является наличие ограничения (и притом нетривиального) на искомую модель. Алгоритм решает эту проблему, переводя формулу ϕ_t на другой язык первого порядка $\mathcal{L}_m$, для которого уже может быть применен решатель. Перевод осуществляется в несколько шагов: (1) замыкание множества типов, (2) расширение отношения подтипирования, (3) формирование аксиом частичного порядка, (4) запрет множественного наследования, (5) формирование ограничений на запечатанные типы, (6) формирование ограничений на ссылочность, (7) формирование ограничений на конструкторы без параметров, (8) формирование ограничений на типовые параметры и (9) формирование окончательной формулы. Шаг 2 предполагает наличие предиката-оракула $groundSubtype$, который для пары замкнутых типов определяет, является ли первый подтипом второго, используя таблицу классов CT.

Далее будет дано общее описание языка $\mathcal{L}_m$ и детальное описание каждого шага алгоритма, проанализирована разрешимость выполнимости формул языка $\mathcal{L}_m$, выдаваемых алгоритмом, а также представлен способ преобразования модели $\mathcal{M}_m$ обратно в модель $\mathcal{M}_t$.

Язык $\mathcal{L}_m$

Язык $\mathcal{L}_m$ описывается сигнатурой $(\varnothing, \mathcal{P}_m)$, где
$\mathcal{P}_m = \{<:, subclass, con, interface, hplc\}$. Здесь $<:$, $subclass$, con — двухместные предикатные символы, а $interface$ и $hplc$ — одноместные. За $\mathcal{V}$ обозначим счетное множество предметных переменных, в идентификаторах которых разрешим использовать «<» и «>».

Замыкание множества типов

Определение 3. Множество типов S будем называть замкнутым относительно декомпозиции, наследования и ограничений (далее просто *за-*

мкнутым), если (a) $C\texttt{<}\overline{T}\texttt{>} \in S \Rightarrow \forall i, T_i \in S$, (b) $T \in S \wedge T <:: V_1, \ldots, V_m \Rightarrow \forall i, V_i \in S$, и (c) $X \in S \Rightarrow \forall i, U_i \in S$, где U_i — элементы правой части в записи таблицы классов $X <: U_1, \ldots, U_n$. *Замыканием* множества S (обозначается $cl(S)$) будем называть минимальное замкнутое множество, содержащее S.

Теорема 1**.** *Для нерасширяющийся таблицы классов замыкание конечного множества типов конечно.*

Доказательство. В доказательстве Теоремы 9 в работе [11] показан аналогичный факт для замыкания относительно декомпозиции и наследования (т.е. без учета п.(c) Определения 3). Обозначим за $cl^{a,b}(A)$ замыкание A относительно декомпозиции и наследования.

Пусть теперь дано конечное множество S. Заметим, что взятие замыкания $cl(S)$ аналогично выполнению двух шагов до достижения неподвижной точки:

1. вычисление $S' = cl^{a,b}(S)$;
2. вычисление нового $S = S' \cup U$, где U — множество всех элементов всех правых частей записей таблицы классов вида $X <: U_1, \ldots, U_n$, где $X \in S'$.

Итак, если S — конечное множество, то S' — также конечное множество, что влечет конечность нового множества S (к S' добавляется конечное множество записей U, т.к. количество типовых переменных в S' конечно). Наконец, заметим, что каждая итерация алгоритма добавляет к S лишь типовые переменные, явно содержащиеся в таблице классов. Но т.к. таблица классов конечна, количество итераций также конечно, что и дает нам конечность $cl(S)$. □

Определим множество S', как множество всех термов, входящих в формулу языка $\mathcal{L}_t$.

Далее надо определить функцию $newVars$, зависящую от таблицы классов CT и натурального числа h. Для этого введём новые обозначения:

- CS — множество не нульарных конструкторов из таблицы классов;

- R — максимальная арность конструкторов из таблицы классов CT;

- $newVar(C, n)$ — функция, возвращающая множество типов, определенная следующим образом:

$$newVar(C, n) \stackrel{\text{def}}{=} \left\{C\langle\overline{X^i}\rangle\right\}_{i=1}^{n},$$

  где все типовые переменные X_j^i различны;

- $count(h,\ R)$ — функция, возвращающая максимальное число раз, которое можно использовать один и тот же конструктор арности R из множества CS для построения типа высоты h:

$$count(h,\ R) \stackrel{\text{def}}{=} \begin{cases} 0, & h = 1 \text{ и } R \neq 0 \\ 1, & h = 1 \text{ и } R = 0 \\ \frac{R^h - 1}{R - 1}, & otherwise \end{cases}$$

Теперь можно определить функцию $newVars$ и результат её применения к выходным данным CT и h:

$$newVars(CT,\ h) \stackrel{\text{def}}{=} \bigcup_{C \in CS} newVar(C,\ count(h, R))$$
$$NV \stackrel{\text{def}}{=} newVars(CT,\ h)$$

Обозначим за S объединение множеств S' и NV. Так как множество S будет конечным, по Теореме 1 его замыкание $cl(S)$ также будет конечным. Выберем произвольное инъективное отображение $\tau : cl(S) \to \mathcal{V}$. Обозначим за FV образ $\tau(cl(S))$. Очевидно, τ задает взаимно-однозначное соответствие между $cl(S)$ и FV.

Пусть предикат $inCT(x)$ истинен, если главный конструктор аргумента принадлежит оригинальной таблице классов, и ложен иначе. Другими словами, для $v \in FV$, $inCT(x)$ истинен, если $x = v$ и $\tau^{-1}(v)$ является применением какого-либо функционального символа в $\mathcal{L}_t$, и

ложен в ином случае[1]:

$$inCT(x) \stackrel{\text{def}}{=} \bigvee_{\substack{T \in cl(S), \\ T \text{ не переменная}}} x = \tau(T)$$

Расширение отношение подтипирования

После построения замыкания множества типов $cl(S)$ и сопоставления им переменных FV алгоритм распространяет отношение подтипирования на пары типов в $cl(S)$, вводя предикат

$$st \stackrel{\text{def}}{=} \bigwedge_{(A,B) \in cl(S) \times cl(S), A \not\equiv B} st_{A,B},$$

где $st_{A,B}$ определяется одним из четырех способов.

- Если и A, и B — закрытые типы, то решение об их подтипировании принимает оракул $groundSubtype$:

$$st_{A,B} \stackrel{\text{def}}{=} \tau(A) <: \tau(B) \Leftrightarrow groundSubtype(A, B)$$

- A или B открыт, но ни один из них не является типовой переменной. В таком случае можно записать ограничения в виде эквивалентных преобразований, которые задаются правилами вывода Var и Super. Введем вспомогательное множество $supertype(C\langle\overline{T}\rangle, D\langle\overline{U}\rangle) = \{D\langle\overline{W}\rangle \mid C\langle\overline{T}\rangle <::^{+} D\langle\overline{W}\rangle\}$. Тогда

$$\tau(C\langle\overline{T}\rangle) <: \tau(D\langle\overline{U}\rangle) \Leftrightarrow \bigvee_{V \in supertype(C\langle\overline{T}\rangle, D\langle\overline{U}\rangle)} \tau(V) <: \tau(D\langle\overline{U}\rangle) \qquad (2)$$

  отображает применение детерминированной версии правила Super, а

$$\tau(C\langle\overline{T}\rangle) <: \tau(C\langle\overline{U}\rangle) \Leftrightarrow \bigwedge_{i} \tau(T_i) <:_{var(C\#i)} \tau(U_i) \qquad (3)$$

  отображает применение правила Var. $st_{A,B}$ в этом случае является

[1] Фактически, такое определение не гарантирует, что $inCT$ опишет *все* конструкторы из таблицы классов CT, а только те, что встретились в замыкании S. Однако, не умаляя общности, мы можем считать, что CT содержит только релевантные для ϕ_t записи

конъюнкцией формул (2) и (3).

В процессе применения правил вывода может произойти зацикливание (*occurs check*). В тексте спецификации CLI [4] отсутствует явное указание на поведение среды в ситуации occurs check, однако в разделе `I.8.7.1` сказано, что отношение подтипирования есть *наименьшее* отношение, замкнутое относительно набора некоторых правил. Из этого следует, что ситуация occurs check отвергает подтипирование одного типа другим. Итак, в случае зацикливания вывода добавляется правило

$$st_{A,B} \stackrel{\text{def}}{=} C\texttt{<}\overline{T}\texttt{>} \not<: D\texttt{<}\overline{U}\texttt{>}$$

- Если $A = C\texttt{<}\overline{T}\texttt{>}$ — сконструированный тип, а $B = X$ — типовая переменная, то необходимо ввести ограничение, показывающее, что типовая переменная X должна принадлежать к изначальной таблице классов CT (в противном случае, решатель может «вклинить» новый класс в иерархию $C\texttt{<}\overline{T}\texttt{>}$). Один из способов сделать это — добавить правило следующего вида:

$$st_{A,B} \stackrel{\text{def}}{=} C\texttt{<}\overline{T}\texttt{>} <: X \Rightarrow inCT(X)$$

- Наконец, случай, когда A — типовая переменная, будет покрыт на последующих шагах. На данном шаге все такие пары игнорируются: $st_{A,B} \stackrel{\text{def}}{=} \top$.

Аксиомы частичного порядка

Исходя из того, что отношение подтипирования является рефлексивным и транзитивным [11], а таблица классов ацикличной, можно доказать, что отношение антисимметрично. Важно явным образом добавить аксиомы частичного порядка, так как это гарантирует, что отношение частичного порядка будет выполняться не только для закрытых

типов, но и для открытых.

$$\begin{aligned} po \stackrel{\text{def}}{=} & (\forall x.\, x <: x) \wedge \\ & (\forall x, y.\, x <: y \wedge y <: x \Rightarrow x = y) \wedge \\ & (\forall x, y, z.\, x <: y \wedge y <: z \Rightarrow x <: z) \end{aligned}$$

Запрет множественного наследования

Множественное наследование в .NET разрешено только для интерфейсов. Если какой-то тип является подтипом не-интерфейса, то он не является интерфейсом, с одним исключением: тип `System.Object` является классом, но также является подтипом для интерфейсов. Для отражения этих свойств используются предикатные символы $interface$ и $subclass$. Предикатный символ $interface$ истинен для всех интерфейсов (т.е. типов, аннотированых в CT буквой I) и ложен для не-интерфейсов (т.е. типов, аннотированых в CT буквой C или V).

$$iface \stackrel{\text{def}}{=} \bigwedge_{\substack{T \in cl(S), \\ T \text{ аннотирован «}I\text{»}}} interface\big(\tau(T)\big) \;\wedge \bigwedge_{\substack{T \in cl(S), \\ T \text{ аннотирован «}C\text{» или «}V\text{»}}} \neg interface\big(\tau(T)\big)$$

$$\begin{aligned} mi \stackrel{\text{def}}{=} & \; iface \wedge (\forall x, y, z.\, subclass(x, y) \wedge subclass(x, z) \Rightarrow \\ & \quad subclass(z, y) \vee subclass(y, z)) \wedge \\ & (\forall x, y.\, \neg interface(x) \wedge \neg interface(y) \Rightarrow \\ & \quad (x <: y \Leftrightarrow subclass(x, y))) \wedge \\ & (\forall x, y.\, \neg interface(y) \wedge x <: y \wedge y \neq \texttt{System.Object} \Rightarrow \\ & \quad \neg interface(x)) \end{aligned}$$

Запечатанные типы

На следующем шаге вводится ограничение $seal$ для запрета наследования запечатанного типа:

$$seal \stackrel{\text{def}}{=} \bigwedge_{\substack{T \in cl(S), \\ T \text{ запечатанный}}} \forall x.\, x <: T \Rightarrow x = T \vee inCT(x)$$

Ссылочные типы и типы значений

Для каждого типового параметра можно указать, будет ли он типом значения или ссылочным типом. Это можно сделать, добавив аннотацию V или R соответственно. Наличие такой аннотации в таблице классов гарантирует, что в ней будет запись с конструктором `System.ValueType`. Эти ограничения будут представлены в виде формулы

$$\begin{gathered} vl(x) = x <: \tau(\texttt{System.ValueType}) \wedge \\ x \neq \tau(\texttt{System.ValueType}) \\ vrt \stackrel{\text{def}}{=} \bigwedge_{\substack{T \in cl(S), \\ T \text{ аннотирован } «R»}} \big(\neg vl(\tau(T))\big) \quad \wedge \bigwedge_{\substack{T \in cl(S), \\ T \text{ аннотирован } «V»}} \big(vl(\tau(T))\big) \end{gathered}$$

Конструкторы без параметров

В некоторых случаях существенны ограничения на переменные вида `where X : new()`. К примеру, если для такой переменной известно, что $C\texttt{<}\overline{T}\texttt{>} <: X$, а в иерархии $C\texttt{<}\overline{T}\texttt{>}$ лишь `System.Object` имеет конструктор без параметров, можно заключить, что X является типом `System.Object`. Ограничения на конструктор без параметров кодируются символом $hplc$.

$$plcs \stackrel{\text{def}}{=} \bigwedge_{\substack{T \in cl(S), \\ T \text{ аннотирован } «()»}} hplc\big(\tau(T)\big) \quad \wedge \bigwedge_{\substack{T \in cl(S), \\ T \text{ не аннотирован } «()», \\ T \text{ не типовая переменная}}} \neg hplc\big(\tau(T)\big)$$

Ограничения на типовые параметры

Для каждого формального типового параметра X в таблице есть запись $X <: T_1 \ldots T_n$, которая ограничивает его; добавим такую запись для каждой типовой переменной, которая является аргументом для конструктора типа.

$$constr \stackrel{\text{def}}{=} \bigwedge_{\substack{C\texttt{<}\overline{U}\texttt{>} \in cl(S), \\ X \in cl(S),\, U_i = X}} \big(\tau(X) <: \tau(T_1)\big) \wedge \ldots \wedge \big(\tau(X) <: \tau(T_n)\big)$$

Запрет бесконечных типов

Так как в системе типов .NET у всех типов конечная высота, а текущие аксиомы не запрещают иметь бесконечную высоту, надо явно указать это свойство:

$$ct \stackrel{\text{def}}{=} \bigwedge_{C<\overline{U}>\in cl(S)} \forall x.con(x, \tau(C<\overline{U}>)) \Leftrightarrow \bigvee_i x = \tau(U_i) \vee con(x, \tau(U_i))$$
$$rl \stackrel{\text{def}}{=} \forall x.\neg con(x, x) \wedge \forall x, y.\neg(con(x, y) \wedge con(y, x))$$
$$ft \stackrel{\text{def}}{=} ct \wedge rl$$

Формирование окончательной формулы

Наконец, алгоритм создает окончательную кодировку в $\mathcal{L}_m$. Пусть ϕ_m — формула, полученная из ϕ_t заменой каждого атома $T \stackrel{<:}{=} U$ на атом $\tau(T) \stackrel{<:}{=} \tau(U)$. Тогда результатом является

$$\psi_m \stackrel{\text{def}}{=} po \wedge mi \wedge st \wedge seal \wedge plcs \wedge vrt \wedge constr \wedge ft \wedge \phi_m$$

Разрешимость

Полученный список формул принадлежит разрешимому фрагменту логики первого порядка, который называется *effectively propositional logic* (EPR) [19]. Алгоритм эффективного решения EPR с равенством, использующий подстановочные множества, реализован, к примеру, в SMT-решателе Z3 [18].

Важно, что если у EPR-формулы есть модель, то у нее есть модель с конечным носителем. Другое важное замечание состоит в том, что алгоритмы решения EPR первым шагом обычно сколемизируют переменные под кванторами существования в функциональные константы. Таким образом, в случае выполнимости формулы ϕ_t, от решателя можно получить оценку переменных под кванторами существования.

Интуитивно, разрешимость системы типовых ограничений возможна из-за ограничений на таблицу классов (она должна быть нерасширяемой) и из-за требования на отсутствие кванторов в ϕ_t.

Преобразование модели

Пусть $cl_{\exists}(\psi_m)$ имеет модель $\mathcal{M}_m = (\mathcal{D}_m, \sigma_m)$ с конечным носителем $(\mathcal{D}_m)$. Более того, как сказано в конце предыдущей секции, вместе с моделью можно получить оценку $val_m : FV \to \mathcal{D}_m$ переменных под кванторами существования. Данная секция обсуждает, как перейти от $\mathcal{M}_m$ к модели $\mathcal{M}_t = (\mathcal{D}_t, \sigma_t)$ формулы $cl_{\exists}(\phi_t)$ в $\mathcal{L}_t$.

Лемма 1. *Если для некоторого* $a \in \mathcal{D}_m$, $\tau(C\langle\overline{T}\rangle) \in val_m^{-1}(\{a\})$ *и* $\tau(D\langle\overline{U}\rangle) \in val_m^{-1}(\{a\})$, *то конструкторы* C *и* D *совпадают и* $\forall i. val_m(\tau(T_i)) = val_m(\tau(U_i))$.

Утверждение Леммы 1 можно вывести из антисимметричности подтипирования путем применения правил (2) и (3) к условиям леммы.

Лемма 2. *Если существуют сконструированный тип* $C\langle\overline{T}\rangle$ *и тип* V *такие, что* $\exists i. val_m(\tau(T_i)) = val_m(\tau(V)$, *тогда* $val_m(\tau(C\langle\overline{T}\rangle)) \neq val_m(\tau(V)$

Утверждение Леммы 2 можно вывести из набора аксиом 4.

Лемма 1 и Лемма 2 дают способ конструирования отображения $d : \mathcal{D}_m \to \mathcal{D}_t$, сопоставляющее элементы носителя $\mathcal{M}_m$ замкнутым типам.

Возьмем отображение $fresh : \mathcal{D}_m \to \mathcal{D}_t$, сопоставляющее элементу из $\mathcal{D}_m$ произвольный нульарный (и потому замкнутый) конструктор типа, отсутствующий в таблице классов CT. Обозначим за $fv(\phi_t)$ множество свободных типовых переменных ϕ_t. Теперь можно индуктивно определить отображение d.

$$d(a) \stackrel{\text{def}}{=} \begin{cases} fresh(a), & \text{если } \tau^{-1}\big(val_m^{-1}(\{a\})\big) \subseteq fv(\phi_t) \\ C\langle\overline{d(T)}\rangle, & \text{если } \tau(C\langle\overline{T}\rangle) \in val_m^{-1}(\{a\}) \end{cases}$$

По Лемме 1 отображение d определено корректно, а по лемме 2 является определенным на всём $\mathcal{D}_m$.

Отображение $val_t \stackrel{\text{def}}{=} d \circ val_m \circ \tau$ является оценкой на $fv(\phi_t)$. Модель же $\mathcal{M}_t$ целиком определяется расширением таблицы классов CT'. Для построения CT' сначала зафиксируем множество новых типов, введенных отображением $fresh$ из носителя $\mathcal{D}_m$:

$$N \stackrel{\text{def}}{=} \big\{ fresh(a) \mid \tau^{-1}\big(val_m^{-1}(\{a\})\big) \subseteq fv(\phi_t) \big\}$$

Теперь построим «каркасы» отношения подтипирования для элементов N, транзитивное замыкание которых даст всю их иерархию. Для этого для каждого $n \in N$ применим алгоритм алгоритм выделения транзитивного остова [15]. Обозначим за B_n множество вершин, соединенных в графе-каркасе с n ребром $(n, \cdot)$. Таблица классов CT' получается из CT добавлением записи $fresh(n)^* <:: d(b_n^1), \ldots, d(b_n^k)$ для каждого $n \in N$. Здесь $b_n^1, \ldots, b_n^k$ — элементы множества B_n. Аннотации для $fresh(n)$ определяются очевидным образом из интерпретаций одноместных предикатных символов языка $\mathcal{L}_m$.

Полученная таблица классов CT' и отображение val_t позволяют построить модель формулы $cl_\exists(\phi_t)$, пригодную для статического анализатора.

Описанный алгоритм является конструктивным доказательством следующей теоремы:

Теорема *2.* $\exists\ h \in \mathbb{N}$ *и семейство формул в пренексной нормальной форме без кванторов существования* $\{\psi_m^i\}_{i=0}^\infty \subset \mathcal{L}_m$ *таких, что выполняются следующие утверждения*

- *Если* $cl_\exists(\psi_m^h)$ *имеет модель, тогда* $cl_\exists(\phi_t)$ *также имеет модель*
- *Если* $cl_\exists(\phi_t)$ *имеет модель,* val_t *— оценка переменных* $fv(\phi_t)$ *и* $\max\limits_{t \in val_m(fv(\phi_t))} height(t) \leqslant h$*, тогда* $cl_\exists(\psi_m^h)$ *имеет модель*

Реализация

Разработанный алгоритм был реализован на языке C#. В качестве входных формул языка $\mathcal{L}_t$ используются условия пути, полученные из символьного интерпретатора, для построения формул языка $\mathcal{L}_m$ использовалось Z3 .NET API. Полученный алгоритм является решателем бескванторных формул в теории .NET типов, и его планируется выложить как расширение Z3 .NET API.

7. Тестирование

Так как реализация необходимой функциональности затронула разные части системы символьного исполнения, чтобы протестировать функциональность всей подсистемы, были написаны интеграционные тесты. Сами тесты является методами, написанными на языке C#, которые необходимо было исполнить символьному интерпретатору. Тестировалось исполнение вызовов обобщенных методов, методов от аргументов с полиморфными типами, кода с операторами is, as и преобразования типа. Реализованная функциональность, вместе с тестами, была влита в основную ветку проекта `V#`.

Заключение

В данной работе были выполнены следующие задачи.

- Спроектирована архитектура подсистемы, отвечающей за интерпретацию типов.
- Реализовано символьное исполнение функций с учетом открытых типов и полиморфизма. Неопределенность динамических типов выражена в символьных типах, на которые накладываются ограничения в виде символьных булевых термов. Поддержка исполнения кода с неопределенными типами реализована с учетом композициональности исполнения.
- Предложен и реализован алгоритм, который решает ситемы ограничений на открытые .NET типы. В частности, разработан и реализован алгоритм, который сводит задачу определения выполнимости набора ограничений на открытые типы платформы .NET к разрешимому фрагменту логики первого порядка.
- Подана статья на конференцию SEIM 2018, с последующей публикацией в CEUR (индексируется в базе данных SCOPUS).
- Проведено тестирование функциональности реализованной подсистемы.

Список литературы

[1] Baldoni R. et al. A survey of symbolic execution techniques // arXiv preprint arXiv:1610.00502. — 2016.

[2] Boyer R. S. Elspas B. Levitt K. N. SELECT—a formal system for testing and debugging programs by symbolic execution // ACM SigPlan Notices. — 1975. — Vol. 10, no. 6. — P. 234–245.

[3] Cseppento L. Micskei Z. Evaluating symbolic execution-based test tools // Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on / IEEE. — 2015. — P. 1–10.

[4] Ecma TC39. TG3. Common Language Infrastructure (CLI). Standard ECMA-335, June 2005.

[5] Fraser G. Arcuri A. Evosuite: automatic test suite generation for object-oriented software // Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering / ACM. — 2011. — P. 416–419.

[6] Godefroid P. Compositional dynamic test generation // ACM Sigplan Notices / ACM. — Vol. 42. — 2007. — P. 47–54.

[7] Havelund K. Pressburger T. Model checking java programs using java pathfinder // International Journal on Software Tools for Technology Transfer (STTT). — 2000. — Vol. 2, no. 4. — P. 366–381.

[8] Hoder K. Bjørner N. Generalized Property Directed Reachability // SAT. — 2012. — Vol. 7317. — P. 157–171.

[9] Hojjat H. et al. A verification toolkit for numerical transition systems // International Symposium on Formal Methods / Springer. — 2012. — P. 247–251.

[10] Islam M. Csallner C. Generating test cases for programs that are coded against interfaces and annotations // ACM Transactions on Software

Engineering and Methodology (TOSEM). — 2014. — Vol. 23, no. 3. — P. 21.

[11] Kennedy A. J. Pierce B. C. On decidability of nominal subtyping with variance. — 2006.

[12] King J. C. A new approach to program testing // ACM SIGPLAN Notices / ACM. — Vol. 10. — 1975. — P. 228–233.

[13] King J. C. Symbolic execution and program testing // Communications of the ACM. — 1976. — Vol. 19, no. 7. — P. 385–394.

[14] Komuravelli A. et al. Automatic abstraction in SMT-based unbounded software model checking // International Conference on Computer Aided Verification / Springer. — 2013. — P. 846–862.

[15] La Poutré J. A. van Leeuwen J. Maintenance of transitive closures and transitive reductions of graphs // International Workshop on Graph-Theoretic Concepts in Computer Science / Springer. — 1987. — P. 106–120.

[16] Landman D. Serebrenik A. Vinju J. J. Challenges for static analysis of Java reflection: literature review and empirical study // Proceedings of the 39th International Conference on Software Engineering / IEEE Press. — 2017. — P. 507–518.

[17] Leino K. R. M. Rümmer P. The Boogie 2 Type System: Design and Verification Condition Generation.

[18] Piskac R. de Moura L. Bjørner N. Deciding effectively propositional logic using DPLL and substitution sets // Journal of Automated Reasoning. — 2010. — Vol. 44, no. 4. — P. 401–424.

[19] Ramsey F. P. On a Problem of Formal Logic // Proceedings of the London Mathematical Society. — 1930. — Vol. 2, no. 1. — P. 264–286.

[20] Schmitt P. H. Ulbrich M. Axiomatization of typed first-order logic // International Symposium on Formal Methods / Springer. — 2015. — P. 470–486.

[21] Tillmann N. De Halleux J. Pex–white box test generation for. net // Tests and Proofs. — 2008. — P. 134–153.